Closures

Closures are a powerful feature in Javascript that allows functions to access and manipulate variables outside their scope. A closure is created when a function is defined inside another function, and the inner function has access to the outer function's variables and parameters.

Here is an example of a closure:

```
function outer() {
  var x = 10;
  function inner() {
    console.log(x);
  }
  return inner;
}
var closure = outer();
closure(); // output: 10
```

In this example, the outer function defines a variable x and a function inner. The inner function has access to x, even though it is defined inside outer. When outer is called and returns inner, we can still access x through the closure.

OOPs Paradigm vs. Functional Programming

The OOPs (Object-Oriented Programming) paradigm and Functional Programming are two different programming styles.

In OOPs, the emphasis is on creating classes that contain data and methods that operate on that data. In contrast, Functional Programming focuses on creating functions that take input and produce output, with no side effects. FP emphasizes the use of immutable data and the avoidance of shared state.

Here is an example code that demonstrates the difference between the two paradigms:

```
// Object-oriented programming (OOP)
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
}

const person = new Person('John', 30);
person.sayHello(); // Output: Hello, my name is John and I'm 30 years old.


// Functional programming (FP)
```

```
const createPerson = (name, age) => {
  const sayHello = () => {
    console.log(`Hello, my name is ${name} and I'm ${age} years old.`);
  };

  return { sayHello };
};

const person = createPerson('John', 30);
person.sayHello(); // Output: Hello, my name is John and I'm 30 years old.
```

In the above code, the Person class represents an object-oriented approach to creating a Person object, where we create a class and define properties and methods on it. On the other hand, the createPerson function represents a functional approach to creating a Person object, where we create a function that returns an object with a sayHello method.

The main difference between OOP and FP is that OOP is focused on defining classes and creating objects that have properties and methods, while FP is focused on defining functions that take inputs and produce outputs without any side effects. OOP emphasizes encapsulation, inheritance, and polymorphism, while FP emphasizes immutability, pure functions, and higher-order functions.

In JavaScript, we can use both OOP and FP paradigms, and in fact, we can even combine the two to create hybrid approaches. The choice of which approach to use depends on the problem we're trying to solve and our personal preferences as developers.

Imperative and Declarative Code Writing

Imperative code is code that specifies how to do something, step by step. Declarative code, on the other hand, specifies what to do, without specifying how to do it. Declarative code is often more concise and easier to read and understand than imperative code.

Here is an example of imperative code:

```
var arr = [1, 2, 3];
for (var i = 0; i < arr.length; i++) {
  arr[i] = arr[i] * 2;
}
console.log(arr); // output: [2, 4, 6]
```

And here is the equivalent declarative code:

```
var arr = [1, 2, 3];
arr = arr.map(function(x) { return x * 2; });
console.log(arr); // output: [2, 4, 6]
```

Mutability and Immutability

In Javascript, mutability refers to the ability to change an object's properties or elements after it has been created. Immutability, in contrast, means that an object's properties or elements cannot be changed after it has been created.

Mutable objects include arrays and objects, while primitive data types like strings and numbers are immutable.

Here is an example of mutable and immutable objects:

```
// mutable object (array)
var myArr = [1, 2, 3];
myArr.push(4);
console.log(myArr); // output: [1, 2, 3, 4]

// immutable object (string)
var myStr = "hello";
myStr.toUpperCase();
console.log(myStr); // output: "hello"
```

In the above example, the array myArr is mutable and can be modified using the push method. In contrast, the string myStr is immutable and cannot be changed, even when using a method like toUpperCase.

Higher-Order Functions and Polyfills

Higher-order functions are functions that take other functions as parameters or return functions as output. They are a powerful tool in functional programming, allowing for code reuse and modularity.

Polyfills are code snippets that provide functionality that may not be available in older browsers. They are often used to add support for newer features of Javascript that are not yet widely supported.

Here is an example of a higher-order function and a polyfill:

```
// higher-order function
function multiplyBy(num) {
  return function(x) {
    return x * num;
  };
}

var double = multiplyBy(2);
console.log(double(5)); // output: 10

// polyfill
if (!Array.prototype.map) {
  Array.prototype.map = function(callback) {
    var arr = [];
    for (var i = 0; i < this.length; i++) {
      arr.push(callback(this[i], i, this));
    }
    return arr;
  };
}

var arr = [1, 2, 3];
var doubled = arr.map(function(x) { return x * 2; });
console.log(doubled); // output: [2, 4, 6]
```

In the above example, the multiplyBy function is a higher-order function that returns a new function that multiplies its input by a given number. The polyfill for the map method checks if the method already exists and, if not, adds it to the Array.prototype object.

Javascript Engines [Article]

Javascript engines are programs that execute Javascript code. They are responsible for interpreting Javascript code, optimizing its performance, and executing it on the user's computer or device.

Some popular Javascript engines include V8 (used in Google Chrome and Node.js), SpiderMonkey (used in Firefox), and JavaScriptCore (used in Safari).

Prototypes of Map, Filter, and Reduce

Map, filter, and reduce are three common methods used in Javascript for working with arrays.

The map method applies a function to each element in an array and returns a new array with the results. The filter method creates a new array with all elements that pass a test specified by a function. The reduce method reduces an array to a single value by applying a function to each element and accumulating the result.

Here is an example of using these methods:

```
var arr = [1, 2, 3];
var doubled = arr.map(function(x) { return x * 2; });
var even = arr.filter(function(x) { return x % 2 == 0; });
var sum = arr.reduce(function(acc, curr) { return acc + curr; }, 0);

console.log(doubled); // output: [2, 4, 6]
console.log(even); // output: [2]
console.log(sum); // output: 6
```

Polyfills of Map, Filter, and Reduce

Polyfills for map, filter, and reduce are used to provide support for these methods in older browsers that may not have them natively implemented. Here are examples of polyfills for each of these methods:

```
// map polyfill
if (!Array.prototype.map) {
  Array.prototype.map = function(callback) {
    var arr = [];
    for (var i = 0; i < this.length; i++) {
      arr.push(callback(this[i], i, this));
    }
    return arr;
  };
}

// filter polyfill
if (!Array.prototype.filter) {
  Array.prototype.filter = function(callback) {
    var arr = [];
    for (var i = 0; i < this.length; i++) {
      if (callback(this[i], i, this)) {
        arr.push(this[i]);
      }
    }
    return arr;
  };
}

// reduce polyfill
if (!Array.prototype.reduce) {
  Array.prototype.reduce = function(callback, initialValue) {
    var accumulator = (initialValue === undefined) ? undefined : initialValue;
    for (var i = 0; i < this.length; i++) {
      if (accumulator !== undefined) {
        accumulator = callback.call(undefined, accumulator, this[i], i, this);
      } else {
        accumulator = this[i];
      }
    }
    return accumulator;
  };
}
```

In the above examples, each polyfill checks if the corresponding method already exists and, if not, adds it to the Array.prototype object. The implementation of each method follows the same logic as their native counterparts.

Strict Mode and Non-Strict Mode

Strict mode is a feature in Javascript that enforces stricter syntax rules and error handling. When strict mode is enabled, certain actions that would normally be allowed are not allowed, and certain errors that would normally be ignored are treated as exceptions.

Non-strict mode is the default mode in Javascript and does not enforce these stricter rules. However, it is recommended to use strict mode in all Javascript code to ensure better code quality and fewer errors.

Here is an example of using strict mode:

```
"use strict";

function foo() {
  x = 10; // throws ReferenceError in strict mode
}

foo();
```

In the above example, the use of strict mode is indicated by the "use strict"; statement at the beginning of the function. This statement enables strict mode for the entire function and prevents the use of undeclared variables, like x in this case.

The this Keyword in Javascript

The this keyword in Javascript refers to the object that the current code is executing in. It can be used to refer to the current object, to create new objects, or to invoke functions with a specific context.

Here is an example of using the this keyword:

```
var obj = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name + "!");
  }
};

obj.greet(); // output: "Hello, John!"
```

In the above example, the this keyword refers to the obj object, which has a name property and a greet method that uses this to access the name property.

Function Currying

Function currying is a technique in functional programming that involves creating a new function by partially applying an existing function. It means that a function that takes multiple arguments is transformed into a sequence of functions that each takes a single argument. The resulting functions can then be composed to perform the original function.

Here's an example to illustrate function currying in JavaScript:

```
function add(a) {
  return function(b) {
    return a + b;
  }
}

const addTwo = add(2); // returns a function that adds 2 to any number
console.log(addTwo(3)); // Output: 5
```

In this example, the add() function takes an argument a and returns a new function that takes another argument b. The new function returns the sum of a and b. We then call the add() function with the argument 2 to create a new function addTwo that takes a single argument and adds 2 to it.

We can use this technique to create new functions that perform a specific operation by partially applying an existing function. This helps in creating reusable code that can be composed to form more complex functions.

Function currying can also be achieved using the bind() method. Here's an example:

```
function add(a, b) {
  return a + b;
}

const addTwo = add.bind(null, 2); // returns a function that adds 2 to any number
console.log(addTwo(3)); // Output: 5
```

In this example, we use the bind() method to partially apply the add() function by passing 2 as the first argument. The null value is passed as the first argument to the bind() method as we don't need to bind this to any object in this case.

Method Borrowing

Method borrowing is a technique in Javascript that involves using the call or apply method to invoke a method from one object on another object. This allows for code reuse and avoids the need to duplicate code.

Here is an example of using the call method to borrow a method from one object and use it on another object:

```
var obj1 = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name + "!");
  }
};

var obj2 = {
  name: "Jane"
};

obj1.greet.call(obj2); // output: "Hello, Jane!"
```

In the above example, the call method is used to invoke the greet method from obj1 on obj2. The this keyword inside the greet method refers to obj2 instead of obj1, allowing the method to be reused on a different object.

Applying and Bind Function

The apply and bind methods are used to set the context of a function and can be used to create new functions with a specific context.

The apply method is used to call a function with a specific context and arguments, while the bind method is used to create a new function with a specific context that can be called later.

Here is an example of using the apply method:

```
function greet() {
  console.log("Hello, " + this.name + "!");
}
```

```
var obj = {
  name: "John"
};

greet.apply(obj); // output: "Hello, John!"
```

In the above example, the apply method is used to call the greet function with a context of obj, which has a name property. The this keyword inside the greet function refers to obj, allowing the function to output "Hello, John!".

Here is an example of using the bind method:

```
function greet() {
  console.log("Hello, " + this.name + "!");
}

var obj = {
  name: "John"
};

var sayHello = greet.bind(obj);
sayHello(); // output: "Hello, John!"
```

In the above example, the bind method is used to create a new function called sayHello that has a context of obj, which has a name property. The sayHello function can be called later and will output "Hello, John!".

Implementing the Bind Method

Here is an example of implementing the bind method using the Function.prototype object:

```
Function.prototype.myBind = function(context) {
  var self = this;
  return function() {
    return self.apply(context, arguments);
  };
};

function greet() {
  console.log("Hello, " + this.name + "!");
}

var obj = {
  name: "John"
};

var sayHello = greet.myBind(obj);
sayHello(); // output: "Hello, John!"
```

In the above example, a new method called myBind is added to the Function.prototype object. This method takes a single argument context, which is the context to set for the function.

Inside the myBind method, a reference to this is stored in the variable self, which refers to the function that myBind is called on. A new function is then returned that calls self.apply(context, arguments), which sets the context of self to context and passes in any arguments that are passed to the returned function.

The greet function is then used to create a new function called sayHello using the myBind method, which has a context of obj. The sayHello function is called later and outputs "Hello, John!".

Constructor Function and “this” Keyword

In JavaScript, a constructor function is used to create objects of a specific type. The this keyword is used inside a constructor function to refer to the object being created.

Here is an example of creating a constructor function:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

var john = new Person("John", 30);
console.log(john.name); // output: "John"
console.log(john.age); // output: 30
```

In the above example, the Person function is a constructor function that takes two arguments name and age. Inside the Person function, the this keyword is used to set the name and age properties of the object being created.

A new object called john is then created using the new keyword and the Person constructor function. The name and age properties of john are set to "John" and 30 respectively.

Prototype

In JavaScript, a prototype is an object that is associated with a constructor function and is used to share properties and methods among all objects created using that constructor function.

Here is an example of using a prototype to add a method to an object:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");
};

var john = new Person("John", 30);
john.greet(); // output: "Hello, my name is John and I am 30 years old."
```

In the above example, a new method called greet is added to the Person constructor function's prototype. This method uses the this keyword to output a string that includes the name and age properties of the object it is called on.

The greet method is then called on the john object, outputting "Hello, my name is John and I am 30 years old.".

Prototypal Inheritance

Prototypal inheritance is a way of creating objects in JavaScript by inheriting properties and methods from a prototype object. In JavaScript, every object has a prototype object, which acts as a template for creating new objects. When a property or method is accessed on an object, JavaScript first looks for the property or method on the object itself. If it doesn't find it, it then looks for it on the object's prototype object. If it still doesn't find it, it looks on the prototype object's prototype object, and so on, until it reaches the root of the prototype chain.

Here's an example to illustrate prototypal inheritance in JavaScript:

```
// Parent object
const Animal = function(name) {
  this.name = name;
};

Animal.prototype.sayName = function() {
  console.log(`My name is ${this.name}`);
};

// Child object
const Dog = function(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
};

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.sayBreed = function() {
  console.log(`My breed is ${this.breed}`);
};

const myDog = new Dog('Buddy', 'Golden Retriever');
myDog.sayName(); // Output: My name is Buddy
myDog.sayBreed(); // Output: My breed is Golden Retriever
```

In this example, we define a parent object Animal that has a name property and a sayName() method. We then create a child object Dog that inherits from the Animal object using Object.create() method. We also define a sayBreed() method on the Dog object.

When we create a new Dog object using the new keyword and pass in a name and breed, the Animal constructor is called with the name argument using call() method to set the name property on the Dog object. The Dog object then inherits the sayName() method from the Animal prototype object using Object.create() method. Finally, we define a new method sayBreed() on the Dog prototype object.

When we call the sayName() and sayBreed() methods on the myDog object, JavaScript first looks for the methods on the myDog object itself. If it doesn't find them, it then looks for them on the Dog prototype object, and then on the Animal prototype object.

Constructor Functions

In JavaScript, a constructor function is a special function that is used to create objects. When a constructor function is called with the new keyword, it creates a new object and sets the this keyword to refer to the newly created object. The constructor function can then add properties and methods to the object using the this keyword.

Here's an example of a constructor function:

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}

const john = new Person('John', 'Doe', 30);
console.log(john.firstName); // Output: John
```

Creating Objects Using Classes

To create objects using classes, we use the new keyword followed by the class name and any constructor arguments that are required. The constructor function is automatically called when the object is created.

Here's an example of creating objects using a class:

```
class Person {
  constructor(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

const john = new Person('John', 'Doe', 30);
console.log(john.getFullName()); // Output: John Doe
```

In this example, we define a Person class with a constructor that takes three arguments (firstName, lastName, and age). We also define a getFullName() method on the class that returns the full name of the person.

We then create a new Person object called john and call the getFullName() method on it to retrieve the full name of the person.

What is the differences between null, undefined and not defined in javascript

- undefined is a variable that has been declared but hasn't been assigned any value yet.
- null is a value that represents the intentional absence of any object value.
- not defined is a variable that has not been declared yet.

Explain truthy and falsy values

In JavaScript, truthy values are those that evaluate to true when coerced to a boolean, while falsy values are those that evaluate to false. Falsy values include:

- false
- 0
- "" (empty string)
- null
- undefined
- NaN

Explain operators in js

Operators in JavaScript are symbols that represent a specific action to be performed on one or more operands. JavaScript has several types of operators:

- Arithmetic operators, such as +, -, *, /, %, etc., perform mathematical operations on operands.
- Comparison operators, such as ==, !=, ===, !==, >, <, >=, <=, etc., compare two values and return a boolean value.
- Logical operators, such as &&, ||, and !, perform logical operations on boolean values.
- Assignment operators, such as =, +=, -= etc., assign a value to a variable.
- Bitwise operators, such as &, |, ^, ~, etc., perform bitwise operations on operands.

Compare operators with truthy and falsy values

Operators in JavaScript can be used with truthy and falsy values just like any other value. When used with a falsy value, most operators will return a falsy value. For example, the comparison operator == will return false if used to compare a falsy value with a truthy value. However, there are some operators that work differently with truthy and falsy values. For example, the logical OR (||) operator returns the first truthy value it encounters, while the logical AND (&&) operator returns the first falsy value it encounters.

Explain shallow copy

In JavaScript, a shallow copy is a copy of an object that creates a new object with the same properties and values as the original object. However, if any of the properties of the original object are objects themselves, the new object will contain references to those objects, rather than creating new copies of them. This means that changes made to those objects in the original object will also be reflected in the new object.

Here's a JavaScript code to create a shallow copy of an object using the Object.assign() method:

```
const originalObj = {
  a: 1,
  b: [2, 3],
  c: {
    d: 4,
    e: 5
  }
};

const copiedObj = Object.assign({}, originalObj);

console.log(originalObj); // {a: 1, b: Array(2), c: {d: 4, e: 5}}
console.log(copiedObj); // {a: 1, b: Array(2), c: {d: 4, e: 5}}

originalObj.a = 10;
originalObj.b.push(4);
originalObj.c.d = 40;

console.log(originalObj); // {a: 10, b: Array(3), c: {d: 40, e: 5}}
console.log(copiedObj); // {a: 1, b: Array(3), c: {d: 40, e: 5}}
```

In the above code, the Object.assign() method is used to create a shallow copy of the originalObj object. The first argument to Object.assign() is an empty object, which serves as the target for the copy. The second argument is the object to be copied.

When we modify any of the nested objects or arrays in the originalObj object, the changes are reflected in the copied object as well, because they share the same references to those nested objects and arrays.

So, in short, a shallow copy of an object creates a new object with the same properties as the original object, but the values of the properties that are objects or arrays are shared between the original and copied objects.

Explain deep copy

In JavaScript, a deep copy is a copy of an object that creates a new object with the same properties and values as the original object, including any objects that are properties of the original object. This means that changes made to any objects in the original object will not be reflected in the new object.

Here's a JavaScript code to create a deep copy of an object using recursion:

```
function deepCopy(obj) {
  let copy;

  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }

  if (Array.isArray(obj)) {
    copy = [];
    for (let i = 0; i < obj.length; i++) {
      copy[i] = deepCopy(obj[i]);
    }
  } else {
    copy = {};
    for (let key in obj) {
      copy[key] = deepCopy(obj[key]);
    }
  }

  return copy;
}

const originalObj = {
  a: 1,
  b: [2, 3],
  c: {
    d: 4,
    e: 5
  }
};

const copiedObj = deepCopy(originalObj);

console.log(originalObj); // {a: 1, b: Array(2), c: {d: 4, e: 5}}
console.log(copiedObj); // {a: 1, b: Array(2), c: {d: 4, e: 5}}

originalObj.a = 10;
originalObj.b.push(4);
originalObj.c.d = 40;

console.log(originalObj); // {a: 10, b: Array(3), c: {d: 40, e: 5}}
console.log(copiedObj); // {a: 1, b: Array(2), c: {d: 4, e: 5}}
```

Explain flattening an object

Flattening an object means transforming a nested object into a flat object. For example, if we have an object like this:

```
const obj = {
  a: {
    b: {
      c: 1,
      d: 2
    },
    e: 3
  },
  f: 4
};
```

we can flatten it into a flat object like this:

```
const flattenedObj = {
  'a.b.c': 1,
  'a.b.d': 2,
  'a.e': 3,
  f: 4
};
```

Code for above conversion is shown below :

```
function flattenObj(obj, prefix = '') {
  const flattened = {};

  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      const newKey = prefix ? `${prefix}.${key}` : key;
      if (typeof obj[key] === 'object' && obj[key] !== null) {
        Object.assign(flattened, flattenObj(obj[key], newKey));
      } else {
        flattened[newKey] = obj[key];
      }
    }
  }

  return flattened;
}

const obj = {
  a: {
    b: {
      c: 1,
      d: 2
    },
    e: 3
  },
  f: 4
};

const flattenedObj = flattenObj(obj);
console.log(flattenedObj);
```